COMP354

Software Engineering

Lecture 15–18

Formal Specifications

Formal Specifications Overview

Aim: to be able to **read** formal specifications

Z — a model-based specification

- system = state space + operations
- schema, Delta schema, Xi schema
- invariants, pre- and post-conditions
- notations: ’ ? !
- deriving properties
- error states and combining schemas
- implementation/refinement

Larch — an algebraic specification

- abstract data types
  =operations and their behaviour
- jargon:
  - sort, defines type for parameters and results
  - signature (=syntax), is operation “prototype”
  - equations (=semantics), define constraints on operations
- generic ADT’s
- equations as axioms, conditional equations
- deriving properties via proof: rewriting, induction

Summary of formal specification

Z Specification Language

Z is a **model-based** specification language

Z separates system into *static* and *dynamic* aspects

static part is a description of the **state space**

a **schema** describes the *components* of a state space and the *invariant* properties of states

dynamic part is described in terms of operations

operations change state and/or map input to output

an operation is described by a **Delta schema** or **Xi schema**

these schemas describe the *relation of inputs to outputs*

and *changes of state*

Z uses mathematical data types — sets, sequences, map

Z uses predicate logic

Z Example: The Birthday Book

1. introduce basic types

   $[NAME, DATE]$

2. define state space by a schema
   - name of schema: *BirthdayBook*
   - components: *known*, *birthday*
   - types of values of components
     - a set of NAME
     - a partial function from NAME to DATE
   - state invariant
     a birthdate is recorded for each known name, and only those names

$$
\begin{array}{|l}
\textit{BirthdayBook} \\
\hline
known : \mathbb{P}\, NAME \\
birthday : NAME \pfun DATE \\
\hline
known = \mathrm{dom}\, birthday \\
\hline
\end{array}
$$

Example of a state **satisfying** the schema

$$known = \{ \mathrm{John}, \mathrm{Mike}, \mathrm{Susan} \}$$

$$
\begin{aligned}
birthday = \{ & \mathrm{John} \mapsto 25\text{–}\mathrm{Mar}, \\
& \mathrm{Mike} \mapsto 20\text{–}\mathrm{Dec}, \\
& \mathrm{Susan} \mapsto 20\text{–}\mathrm{Dec} \}.
\end{aligned}
$$

3a. Define operations

An operation to add a birthday to the book

AddBirthday

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$

$$name? \notin known$$

$$birthday' = birthday \cup \{name? \mapsto date?\}$$

$\Delta BirthdayBook$ indicates a possible state change

before state: *known*, *birthday*

after state: $known'$, $birthday'$

NB they satisfy the state invariant of *BirthdayBook*

they also satisfy the predicates in *AddBirthday* schema

pre-condition: *name is not already known*

postcondition: *name is in the birthday book*

? indicates that a variable is an input

3b. Reason about specification:

A sample proof

Can we deduce an expected fact:

$$known' = known \cup \{name?\}.$$

Yes, using the invariants on the state before and after the operation:

$$
\begin{aligned}
& known' \\
&= \mathrm{dom}\, birthday' && \text{[invariant after]} \\
&= \mathrm{dom}(birthday \cup \{name? \mapsto date?\}) && \text{[spec. of } AddBirthday\text{]} \\
&= \mathrm{dom}\, birthday \cup \mathrm{dom}\, \{name? \mapsto date?\} && \text{[fact about dom]} \\
&= \mathrm{dom}\, birthday \cup \{name?\} && \text{[fact about dom]} \\
&= known \cup \{name?\}. && \text{[invariant before]}
\end{aligned}
$$

3a. Another operation

FindBirthday

$\Xi BirthdayBook$
$name? : NAME$
$date! : DATE$

$name? \in known$

$date! = birthday(name?)$

$\Xi BirthdayBook$ indicates no state change, so

$known = known'$

$birthday = birthday'$

! indicates an output variable

3a. Another operation

returns the *set of cards* to send on a particular day

Remind

$$\Xi BirthdayBook$$
$$today? : DATE$$
$$cards! : \mathbb{P}\, NAME$$

$$cards! = \{ n : known \mid birthday(n) = today? \}$$

4. The initial state of the system

InitBirthdayBook

$$BirthdayBook$$

$$known = \emptyset$$

Incremental development: Adding error conditions

5. Add error conditions

5a. An extra output to report errors

declare enumerations by listing the alternatives

$$REPORT ::= ok \mid already\_known \mid not\_known$$

Success

$$result! : REPORT$$

$$result! = ok$$

5b. Add error handling operation

AlreadyKnown

$$\Xi BirthdayBook$$
$$name? : NAME$$
$$result! : REPORT$$

$$name? \in known$$

$$result! = already\_known$$

5c. Define robust version of AddBirthday: compose schemas

$$RAddBirthday \mathrel{\widehat=} (AddBirthday \land Success) \lor AlreadyKnown.$$

Composition of Schemas

Logical connectives combine entire specifications

A **robust specification** handles all cases

$$RAddBirthday \mathrel{\widehat=} (AddBirthday \land Success) \lor AlreadyKnown.$$

The schema *RAddBirthday* in full:

RAddBirthday

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$
$$result! : REPORT$$

$$(name? \notin known \land$$
$$\quad birthday' = birthday \cup \{name? \mapsto date?\} \land$$
$$\quad result! = ok) \lor$$
$$(name? \in known \land$$
$$\quad birthday' = birthday \land$$
$$\quad result! = already\_known)$$

Robust versions of *FindBirthday* and *Remind*

NotKnown

$\Xi BirthdayBook$
$name? : NAME$
$result! : REPORT$

$name? \notin known$

$result! = not\_known$

Robust *FindBirthday* checks that the name is known

$$RFindBirthday \mathrel{\widehat=} (FindBirthday \land Success) \lor NotKnown.$$

The *Remind* operation can be called at any time

$$RRemind \mathrel{\widehat=} Remind \land Success.$$

Refinement of the Specification

A more concrete description of the system using arrays of names and dates.

Remember an array is just a map

$$names : \mathbb{N}_1 \longrightarrow NAME$$
$$dates : \mathbb{N}_1 \longrightarrow DATE.$$

1. Define the more concrete state space
(*hwm* stands for "high water mark")

BirthdayBook1

$$names : \mathbb{N}_1 \longrightarrow NAME$$
$$dates : \mathbb{N}_1 \longrightarrow DATE$$
$$hwm : \mathbb{N}$$

$$\forall i, j : 1 \,..\, hwm \bullet$$
$$\quad i \neq j \Rightarrow names(i) \neq names(j)$$

2. Define the abstraction homorphism relating the two models

Abs

$$BirthdayBook$$
$$BirthdayBook1$$

$$known = \{ i : 1 \,..\, hwm \bullet names(i) \}$$

$$\forall i : 1 \,..\, hwm \bullet$$
$$\quad birthday(names(i)) = dates(i)$$

3. Define concrete operations

To add a new entry, we increment the array index.

$f \oplus \{ x \mapsto y \}$ is the same as f except that $f(x) = y$

AddBirthday1

$\Delta BirthdayBook1$
$name? : NAME$
$date? : DATE$

$\forall i : 1 \,..\, hwm \bullet name? \neq names(i)$

$hwm' = hwm + 1$
$names' = names \oplus \{hwm' \mapsto name?\}$
$dates' = dates \oplus \{hwm' \mapsto date?\}$

We can show:

1. Whenever *AddBirthday* is legal in some abstract state, the implementation *AddBirthday1* is legal in any corresponding concrete state.
2. The final state which results from *AddBirthday1* represents an abstract state which *AddBirthday* could produce.

It is now a simple matter to write code to implement the specification.

```
procedure AddBirthday (Name: NAME; Date: DATE);
  begin
    hwm := hwm + 1;
    names[hwm] := Name;
    dates[hwm] := Date
  end;
```

A search is abstracted by existential quantification.

*FindBirthday*1

$$\Xi BirthdayBook1$$
$$name? : NAME$$
$$date! : DATE$$

$$\exists i : 1 \mathinner{..} hwm \bullet name? = names(i) \land date! = dates(i)$$

```
procedure FindBirthday (Name: NAME; var Date: DATE);
  var i: integer;
  begin
    i := 1;
    while names[i] <> Name do
      i := i + 1;
    Date := dates[i]
  end;
```

Initialization is straightforward:

*InitBirthdayBook*1

$$BirthdayBook1$$

$$hwm = 0$$

Initialization leaves the set of known names empty:

$$\begin{aligned} & known \\ &= \{ i : 1 \mathinner{..} hwm \bullet names(i) \} && \text{[from } Abs\text{]} \\ &= \{ i : 1 \mathinner{..} 0 \bullet names(i) \} && \text{[from } InitBirthdayBook1\text{]} \\ &= \varnothing. && \text{[since } 1 \mathinner{..} 0 = \varnothing\text{]} \end{aligned}$$

```
procedure Initialize;
  begin
    hwm := 0
  end;
```

Summary of Z

precise

separation of concerns

- static vs dynamic
- modularization into schemas
- decomposition of schemas
  eg normal behaviour and error conditions

abstraction homomorphism aids traceability

supports refinement — in several steps, if necessary — from specification to implementation

Larch — Algebraic Specifications

An algebraic specification views an ADT as

an abstract algebra of values

a set of operations that manipulate those values

axioms/rules that spcify the "meaning" of the operation

Larch Shared Language

```
algebra StringSpec
introduces
   sorts String, Char, Nat, Bool;
   operations
      new:     ()              -> String;
      append:  String, String -> String;
      add:     String, Char   -> String;
      length:  String         -> Nat;
      isEmpty: String         -> Bool;
      equal:   String, String -> Bool;
constrains new, append, add, length, isEmpty, equal so that
for all [ s, s1, s2: String; c: Char ]
      isEmpty( new() ) = true;
      isEmpty( add(s,c) ) = false;
      length( new() ) = 0;
      length( add(s,c) ) = length(s) + 1;
      append( s, new() ) = s;
      append( s1, add(s2,c) ) = add( append(s1,s2), c );
      equal( new(), new() ) = true;
      equal( new(), add(s,c) ) = false;
      equal( add(s1,c), new() ) = false;
      equal( add(s1,c), add(s2,c) ) = equal( s1, s2 );
end StringSpec.
```

Larch — Implementing from Algebraic Specifications

Larch Shared Language (LSL)

specify ADTs independent of implementation language

Larch Interface Language, eg Larch/Pascal

connect between LSL ADTs and the implementation

it uses Pascal types
it defines Pascal routines for the operations

specifies interfaces like the MIS document

also includes traceability information by documenting the mapping (called an *abstraction homomorphism*) between the algebraic specification (in LSL) of ADT and the implementation representation (in Pascal)

Larch/Pascal Example

```
type String exports isEmpty, add, append, length
based on Boolean, integer, char

   function isEmpty( s: String ): Boolean
      modifies at most []          {ie it has no side-effects}
      requires true                {ie no precondition, usually omit}
      ensures result = isEmpty(s)  {refers to StringSpec::isEmpty}

   procedure add( var s: String; c: char )
      modifies at most [s]        {ie the only side-effects are
                                        to maybe modify s}
      ensures s' = add( s, c )    {prime ' indicates post-value of s}

   function length( s: String ): integer
      modifies at most []
      ensures result = length(s)

   procedure append( var s1, s2, s3: String )
      modifies at most [s3]       {ie the only side-effects are
                                      to maybe modify s3, although
                                      all of s1, 2, s3 are passed
                                        by reference}
      ensures s3' = append( s1, s2 )

end String.
```

symbols in pre- and post-conditions refer to LSL
they are "abstract" values

′ refers to final value, eg $s3'$

Summary of Formal Specifications

"formal" means the notation used has a precisely defined syntax and semantics

usually notation is based on
logic + set theory + algebra ...

NB A *formal* specification should include
an *informal* English description
to help the user understand the formal specification

operational specification describes behaviour of system
eg finite state machines, petri nets, data flow diagrams

declarative specification describes properties of system
eg logic, Z

NB There is a close interaction between
- requirements analysis and specification
- architectural design
- formal requirements specification

Advantages of Formal Specifications

The development of a formal specification provides insights into and understanding of the software requirements and the software design.

Given a formal specification and a complete formal definition of a programming language, it may be possible to prove that a program conforms to its specification.

automatic processing of formal specifications

basis for tools

animate a formal specification and provide a prototype

Formal specifications are mathematical entities, and can be studied and analysed using mathematical techniques.

can guide the tester to identify test cases

Obstacles to Adoption of Formal Specifications

Management is inherently conservative and unwilling to adopt new techniques whose payoff is not obvious.

Most software engineers have not been trained in techniques of formal software specifications.

System procurers are unwilling to fund development activities they cannot readily influence (because they do not understand the notations etc).

Some classes of software are very difficult to specify

- user interfaces
- interrupt-driven systems
- real-time systems

Widespread ignorance of formal methods and their applicability.

Lack of tools

Poor public relations by academics (in formal methods) who do not understand practical software engineering.

Verification of Formal Specifications

1. by observing dynamic behaviour

   - simulation, using prototypes

   - animation of specifications, using symbolic execution

2. by analysing properties

   eg, deduce properties from the specification

   - by hand

   - using theorem provers